Crate commonware_p2p

Settings

Help

Summary

Source

Communicate with authenticated peers over encrypted connections.

Status

`commonware-p2p` is ALPHA software and is not yet recommended for production use. Developers should expect breaking changes and occasional instability.

Modules

- authenticated
- Communicate with a fixed set of authenticated peers over encrypted connections.
- simulated
- Simulate networking between peers with configurable link behavior (i.e. drops, latency, corruption, etc.).
- utils
- Utility functions for exchanging messages with many peers.

Enums

- Recipients
- Enum indicating the set of recipients to send a message to.

Traits

- Receiver
- Interface for receiving messages from arbitrary recipients.
- Sender
- Interface for sending messages to a set of recipients.

Type Aliases

- Channel
- Alias for identifying communication channels.
- Message
- Tuple representing a message received from a given public key.

Module authenticated

Settings

Help

Summary

Source

Communicate with a fixed set of authenticated peers over encrypted connections.

`authenticated` provides multiplexed communication between fully-connected peers identified by a developer-specified cryptographic identity (i.e. BLS, ed25519, etc.). Peer discovery occurs automatically using ordered bit vectors (sorted by authorized cryptographic identities) to efficiently communicate knowledge of dialable peers.

Features

- Configurable Cryptography Scheme for Peer Identities (BLS, ed25519, etc.)
- Automatic Peer Discovery Using Bit Vectors (Used as Ping/Pongs)
- Multiplexing With Configurable Rate Limiting Per Channel and Send Prioritization
- Optional Message Compression (using `zstd`)

Design

Discovery

Peer discovery relies heavily on the assumption that all peers are known and synchronized at each index (a user-provided tuple of `(u64, Vec<PublicKey>)`). Using this assumption, we can construct a sorted bit vector that represents our knowledge of peer IPs (where 1 == we know, 0 == we don't know). This means we can represent our knowledge of 1000 peers in only 125 bytes!

If peers at a given index are not synchronized, peers may signal their knowledge of peer IPs that another peer may incorrectly respond to (associating a given index with a different peer) or fail to respond to (if the bit vector representation of the set is smaller/larger than expected). It is up to the application to ensure sets are synchronized.

Because this representation is so efficient/small, peers send bit vectors to each other periodically as a "ping" to keep the connection alive. Because it may be useful to be connected to multiple indexes of peers at a given time (i.e. to perform a DKG with a new set of peers), it is possible to configure this crate to maintain connections to multiple indexes (and pings are a random index we are trying to connect to).

```
message BitVec {
    uint64 index = 1;
    bytes bits = 2;
  }
```

Upon receiving a bit vector, a peer will select a random collection of peers (under a configured max) that it knows about that the sender does not. If the sender knows about all peers that we know about, the receiver does nothing (and relies on its bit vector to serve as a pong to keep the connection alive).

```
message Peers {
    repeated Peer peers = 1;
  }
```

If a peer learns about an updated address for a peer, it will update the record it has stored (for itself and for future gossip). This record is created during instantiation and is sent immediately after a connection is established (right after the handshake). This means that a peer that learned about an outdated record for a peer will update it immediately upon being dialed.

```
message Peer {
    bytes socket = 1;
    uint64 timestamp = 2;
    Signature signature = 3;
  }
```

To get all of this started, a peer must first be bootstrapped with a list of known peers/addresses. The peer will dial these other peers, send its own record, send a bit vector (with all 0's except its own position in the sorted list), and then wait for the other peer to respond with some set of unknown peers. Different peers do not need to agree on who this list of bootstrapping peers is (this list is configurable). Knowledge of bootstrappers and connections to them are never dropped, even if the bootstrapper is not in any known peer set.

If a peer is not in any registered peer set (to its knowledge) but is dialed by a peer that is, it will accept the connection. This allows peers that have a more up-to-date version of the

peer set to connect, exchange application-level information, and for the said peer to potentially learn of an updated peer set (of which it is part).

Messages

Messages are sent using the Data message type. This message type is used to send arbitrary bytes to a given channel. The message must be smaller (in bytes) than the configured maximum message size. If the message is larger, an error will be returned. It is possible for a sender to prioritize messages over others.

```
message Data {
    uint32 channel = 1;
    bytes message = 2;
   }
```

Example

```
use commonware_p2p::authenticated::{self, Network};
use commonware_cryptography::{Ed25519, Scheme};
use commonware_runtime::{tokio::{self, Executor}, Spawner,
Runner};
use governor::Quota;
use prometheus_client::registry::Registry;
use std::net::{IpAddr, Ipv4Addr, SocketAddr};
use std::num::NonZeroU32;
use std::sync::{Arc, Mutex};

// Configure runtime
let runtime_cfg = tokio::Config::default();
let (executor, runtime) = Executor::init(runtime_cfg.clone());

// Configure prometheus registry
let registry =
Arc::new(Mutex::new(Registry::with_prefix("p2p")));

// Generate identity
//
// In production, the signer should be generated from a secure
source of entropy.
let signer = Ed25519::from_seed(0);

// Generate peers
//
```

```
// In production, peer identities will be provided by some
external source of truth
// (like the staking set of a blockchain).
let peer1 = Ed25519::from_seed(1).public_key();
let peer2 = Ed25519::from_seed(2).public_key();
let peer3 = Ed25519::from_seed(3).public_key();

// Configure bootstrappers
//
// In production, it is likely that the address of
bootstrappers will be some public address.
let bootstrappers = vec![(peer1.clone(),
SocketAddr::new(IpAddr::V4(Ipv4Addr::LOCALHOST), 3001))];

// Configure namespace
//
// In production, use a unique application namespace to
prevent cryptographic replay attacks.
let application_namespace = b"my-app-namespace";

// Configure network
//
// In production, use a more conservative configuration like
`Config::recommended`.
const MAX_MESSAGE_SIZE: usize = 1_024; // 1KB
let p2p_cfg = authenticated::Config::aggressive(
    signer.clone(),
    application_namespace,
    registry,
    SocketAddr::new(IpAddr::V4(Ipv4Addr::LOCALHOST), 3000),
    bootstrappers,
    MAX_MESSAGE_SIZE,
);

// Start runtime
executor.start(async move {
    // Initialize network
    let (mut network, mut oracle) =
Network::new(runtime.clone(), p2p_cfg);

    // Register authorized peers
    //
    // In production, this would be updated as new peer sets
are created (like when
```

```
    // the composition of a validator set changes).
    oracle.register(0, vec![signer.public_key(), peer1, peer2,
peer3]);

    // Register some channel
    const MAX_MESSAGE_BACKLOG: usize = 128;
    const COMPRESSION_LEVEL: Option<u8> = Some(3);
    let (sender, receiver) = network.register(
        0,
        Quota::per_second(NonZeroU32::new(1).unwrap()),
        MAX_MESSAGE_BACKLOG,
        COMPRESSION_LEVEL,
    );

    // Run network
    let network_handler = runtime.spawn("network",
network.run());

    // ... Use sender and receiver ...

    // Shutdown network
    network_handler.abort();
  });
```

Structs

- Config
- Configuration for the peer-to-peer instance.
- Network
- Implementation of an `authenticated` network.
- Oracle
- Mechanism to register authorized peers.
- Receiver
- Channel to asynchronously receive messages from a channel.
- Sender
- Sender is the mechanism used to send arbitrary bytes to a set of recipients over a pre-defined channel.

Enums

- Error
- Errors that can occur when interacting with the network.

Type Aliases

- Bootstrapper
- Known peer and its accompanying address that will be dialed on startup.

Module simulated

Settings

Help

Summary

Source

Simulate networking between peers with configurable link behavior (i.e. drops, latency, corruption, etc.).

Both peer and link modification can be performed dynamically over the lifetime of the simulated network. This can be used to mimic transient network partitions, offline nodes (that later connect), and/or degrading link quality.

Determinism

`commonware-p2p::simulated` can be run deterministically when paired with `commonware-runtime::deterministic`. This makes it possible to reproduce an arbitrary order of delivered/dropped messages with a given seed.

Example

```
use commonware_p2p::simulated::{Config, Link, Network};
use commonware_cryptography::{Ed25519, Scheme};
use commonware_runtime::{deterministic::Executor, Spawner,
Runner};
use prometheus_client::registry::Registry;
use std::sync::{Arc, Mutex};

// Generate peers
let peers = vec![
    Ed25519::from_seed(0).public_key(),
    Ed25519::from_seed(1).public_key(),
    Ed25519::from_seed(2).public_key(),
```

```
    Ed25519::from_seed(3).public_key(),
];

// Configure network
let p2p_cfg = Config {
    registry:
Arc::new(Mutex::new(Registry::with_prefix("p2p"))),
};

// Start runtime
let (executor, runtime, _) = Executor::seeded(0);
executor.start(async move {
    // Initialize network
    let (network, mut oracle) = Network::new(runtime.clone(),
p2p_cfg);

    // Start network
    let network_handler = runtime.spawn("network",
network.run());

    // Link 2 peers
    oracle.add_link(
        peers[0].clone(),
        peers[1].clone(),
        Link {
            latency: 5.0,
            jitter: 2.5,
            success_rate: 0.75,
        },
    ).await.unwrap();

    // Register some channel
    let (sender, receiver) = oracle.register(
        peers[0].clone(),
        0,
        1024 * 1024, // 1MB
    ).await.unwrap();

    // ... Use sender and receiver ...

    // Update link
    oracle.add_link(
        peers[0].clone(),
        peers[1].clone(),
```

```
        Link {
            latency: 100.0,
            jitter: 25.0,
            success_rate: 0.8,
        },
    ).await.unwrap();

    // ... Use sender and receiver ...

    // Shutdown network
    network_handler.abort();
});
```

Structs

- Config
- Configuration for the simulated network.
- Link
- Describes a connection between two peers.
- Network
- Implementation of a simulated network.
- Oracle
- Interface for modifying the simulated network.
- Receiver
- Implementation of a `crate::Receiver` for the simulated network.
- Sender
- Implementation of a `crate::Sender` for the simulated network.

Enums

- Error
- Errors that can occur when interacting with the network.

commonware_p2p

Module utils Copy item path

Settings

Help

Summary

Source

Utility functions for exchanging messages with many peers.

Modules

- requester
- Make concurrent requests to peers limited by rate and prioritized by performance.

Enum Recipients

Copy item path

Settings

Help

Summary

Source

```
pub enum Recipients {
    All,
    Some(Vec<PublicKey>),
    One(PublicKey),
}
```

Enum indicating the set of recipients to send a message to.

Variants

`All`

`Some(Vec<PublicKey>)`

`One(PublicKey)`

Trait Implementations

Source

`impl Clone for Recipients`

Source

`fn clone(&self) -> Recipients`

Returns a copy of the value. Read more

1.0.0 · Source

```
fn clone_from(&mut self, source: &Self)
```

Performs copy-assignment from `source`. Read more

Auto Trait Implementations

```
impl !Freeze for Recipients
```

```
impl RefUnwindSafe for Recipients
```

```
impl Send for Recipients
```

```
impl Sync for Recipients
```

```
impl Unpin for Recipients
```

```
impl UnwindSafe for Recipients
```

Blanket Implementations

Source

```
impl<T> Any for T

where
    T: 'static + ?Sized,
```

Source

```
impl<T> Borrow<T> for T

where
    T: ?Sized,
```

Source

```
impl<T> BorrowMut<T> for T

where
    T: ?Sized,
```

Source

```
impl<T> CloneToUninit for T

where
    T: Clone,
```

Source

```
impl<T> Conv for T
```

Source

```
impl<T> FmtForward for T
```

Source

```
impl<T> From<T> for T
```

Source

```
impl<T> Instrument for T
```

Source

```
impl<T, U> Into<U> for T

where
    U: From<T>,
```

Source

```
impl<T> IntoEither for T
```

Source

```
impl<T> Pipe for T

where
    T: ?Sized,
```

Source

```
impl<T> Pointable for T
```

Source

```
impl<T> Same for T
```

Source

```
impl<T> Tap for T
```

Source

```
impl<T> ToOwned for T

where
    T: Clone,
```

Source

```
impl<T> TryConv for T
```

Source

```
impl<T, U> TryFrom<U> for T

where
    U: Into<T>,
```

Source

```
impl<T, U> TryInto<U> for T

where
    U: TryFrom<T>,
```

Source

```
impl<V, T> VZip<V> for T

where
    V: MultiLane<T>,
```

Source

impl<T> WithSubscriber for T

commonware_p2p

Trait Receiver

Settings

Help

Summary

Source

```
pub trait Receiver:
    Debug
    + Send
    + 'static {
    type Error: Debug + StdError + Send;

    // Required method
    fn recv(
        &mut self,
    ) -> impl Future<Output = Result<Message, Self::Error>> +
Send;
}
```

Interface for receiving messages from arbitrary recipients.

Required Associated Types

Source

type Error: Debug + StdError + Send

Required Methods

Source

fn recv(&mut self) -> impl Future<Output = Result<Message, Self::Error>> + Send

Receive a message from an arbitrary recipient.

Dyn Compatibility

§

This trait is not dyn compatible.

In older versions of Rust, dyn compatibility was called "object safety", so this trait is not object safe.

Implementors

Source

impl Receiver for commonware_p2p::authenticated::Receiver

Source

impl Receiver for commonware_p2p::simulated::Receiver

commonware_p2p

Trait Sender

Settings

Help

Summary

Source

```
pub trait Sender:
    Clone
    + Debug
    + Send
    + 'static {
    type Error: Debug + StdError + Send;

    // Required method
    fn send(
        &mut self,
        recipients: Recipients,
        message: Bytes,
        priority: bool,
    ) -> impl Future<Output = Result<Vec<PublicKey>, Self::Error>>
+ Send;
}
```

Interface for sending messages to a set of recipients.

Required Associated Types

Source

```
type Error: Debug + StdError + Send
```

Required Methods

Source

```
fn send(
    &mut self,
    recipients: Recipients,
    message: Bytes,
    priority: bool,
) -> impl Future<Output = Result<Vec<PublicKey>, Self::Error>> +
Send
```

Send a message to a set of recipients.

Dyn Compatibility

This trait is not dyn compatible.

In older versions of Rust, dyn compatibility was called "object safety", so this trait is not object safe.

Implementors

Source

impl Sender for commonware_p2p::authenticated::Sender

Source

impl Sender for commonware_p2p::simulated::Sender